

BOOTSTRAP, PC BIOS, AND IA32 MEMORY MODES

CS124 – Operating Systems
Winter 2015-2016, Lecture 5

Bootstrapping

- All computers have the same basic issue:
 - They require a program to tell them what to do...
 - ...but when you turn them on, they have no program!
 - How do you get them to start running programs?
- Computers must implement a **bootstrap process** to load the operating system
 - A series of one or more boot-loaders
 - Each stage is responsible for loading the next stage of the computer's programming
- Originated from the term “pulling oneself up by one's bootstraps”
 - Was used in the context of computing in the early 1950s
 - (The phrase was extant well before that time)

Bootstrapping (2)

- Modern computers typically use read-only memory (ROM) containing the initial code to load the operating system
 - Processors are often implemented to start executing code at a specific address, immediately after they are reset
- Pre-ROM computers had to implement various techniques
 - Read a limited number of instructions from some external source, and then begin executing them!
- Computers with punched-card readers often read one card and then began executing that card's instructions
 - The card would contain a program loader that read subsequent cards, then begin executing the program on those cards

Bootstrapping (3)

- Some computers included a bank of switches to specify the first instruction(s) for the computer to execute
 - e.g. IBM 650 (ca. 1953) had a bank of ten 10-digit switches to specify the first instruction, which was usually “load a card to a specific address, then begin executing the code at that address”
 - e.g. CDC 6600 (ca. 1964) had a bank of 12x12 toggle switches called a dead start panel, which specified first 12 instructions to run
- This process was used even in early microcomputers
 - e.g. MITS Altair 8800 (ca. 1975)
 - Computer had a bank of switches for entering the bootloader’s instructions one by one
 - Once the bootloader was entered, another switch would start the CPU executing the bootloader

Bootstrapping (4)

- Other computers used diode matrixes
 - (e.g. GE 645, ca. 1965)
 - Presence of a diode = 1 bit, absence of a diode = 0 bit
 - Allowed somewhat more complex bootloaders to be specified
- The advent of read-only memory (ROM) revolutionized boot-loading
 - Manufacturers included sophisticated programs on the computer motherboard to manage the operating system boot process
 - Software even includes basic drivers for disks, video, minimal operating systems, etc.
- Combination of persistent memory and the program stored in it is called **firmware**
 - Later, electrically erasable versions of ROM (EPROM, EEPROM) allowed this software to be upgraded

Bootstrapping (5)

- Even back in the day, Apple was focused on building the most user-friendly computers
- Excerpts from a 1976 ad for the Apple 1 computer

You Don't Need an Expensive Teletype.

Using the built-in video terminal and keyboard interface, you avoid all the expense, noise and maintenance associated with a teletype. And the Apple video terminal is six times faster than a teletype, which means more throughput and less waiting.

No More Switches, No More Lights.

Compared to switches and LED's, a video terminal can display vast amounts of information simultaneously. The Apple video terminal can display the contents of 192 memory locations at once on the screen. And the firmware in PROMS enables you to enter, display and debug programs (all in hex) from the keyboard, rendering a front panel unnecessary. The firmware also allows your programs to print characters on the display, and since you'll be looking at letters and numbers instead of just LED's, the door is open to all kinds of alphanumeric software (i.e., Games and BASIC).

Byte into an Apple \$666.66*

* includes 4K bytes RAM

IA32 Bootstrap

- Now: Cover IA32 bootstrap process in all its gory detail
 - Single-core processor bootstrap
- Problem:
 - Operating systems run on hardware, and hardware is complicated
 - Especially true of IA32 architecture, which has remained backward-compatible for decades
- Well-designed operating systems largely obscure these nasty details, even within the OS implementation
 - Unfortunately, you can't completely avoid them, particularly during the bootstrap process

IA32 Bootstrap (2)

- IA32 processors are engineered to start executing instructions at 0xFFFFFFF0 immediately after a reset
- Computer manufacturers place a read-only memory (ROM) in this address range to start the boot process
 - ROM is typically much larger than 16 bytes
 - Modern computers include very sophisticated firmware now
- IA32 CPU also start off in protection level 0 (kernel mode)
 - Allows bootloader to have total access to all system facilities in order to set up the hardware to run the operating system
- Currently there are two categories of PC boot-loaders:
 - PC BIOS – Basic Input/Output System
 - EFI/UEFI – [Unified] Extensible Firmware Interface

PC BIOS

- Original firmware for x86 computers
- Provides two critical features, and a third useful one:
 - A firmware bootloader to start the bootstrap process
 - A library of basic input/output functions for interacting with the computer hardware
 - i.e. device drivers exposed via a standard interface
 - Often includes a simple user interface for hardware configuration
- Originated with the CP/M operating system (1976)
 - At that point the BIOS was actually loaded from disk by firmware
- Functionality emerged as a de-facto standard
 - Certain microcomputers (IBM PC) and operating systems (MS DOS) became extremely popular
 - Other manufacturers began to clone the hardware; they had to match the existing firmware functionality for software to work

PC BIOS Bootloading

- BIOS bootloader follows a very simple process:
- If a hardware reset was performed, run some diagnostics on the hardware
 - e.g. memory check
 - Called a **Power-On Self Test**, a.k.a. POST
- Identify and configure computer peripherals for basic use
- Iterate through bootable devices in some order, trying to load and start the next stage of the bootstrap process
 - Can configure the boot order in current BIOSes
 - Earlier BIOSes simply went through the floppy disks in order, then the hard disks in order
 - The first sector of each bootable device is loaded into memory
 - If sector ends with signature 0x55, 0xAA, it is used as bootloader

PC BIOS Bootloading (2)

- BIOS loads the boot sector at memory address 0x7C00, then jumps to that address
- The boot sector that BIOS loads is only 512 bytes (!!!)
 - Historically, this was the size of x86 disk sectors
 - 0x200 bytes, so bootloader is at addresses 0x7C00 – 0x7DFF
 - Minus the signature, boot sector has 510 bytes to do its thing
- BIOS passes a few limited details to the bootloader
 - e.g. %d1 register contains the numeric ID of the boot disk; allows the bootloader to retrieve more data from the boot disk
- Bootloaders are usually written in assembly language
 - Only way to cram the required functionality into the limited space!

Bootloading and MBRs

- Picture is made more complex by hard disk **partitioning**
- First sector of a hard disk is a **master boot record (MBR)**
 - Specifies up to four **partitions** of the hard disk, each with its own format and use
 - e.g. each partition could be used for a different operating system
 - An operating system might also need multiple partitions, e.g. Linux filesystem partition vs. Linux swap partition
- Issue: MBR doesn't correspond to a specific operating system; the disk may contain multiple OSes!
- A partition containing an operating system often specifies its own bootloader in the first sector of the partition
- MBR must kick off the next bootloader in the process...

Bootloading and MBRs (2)

- Partition details in the master boot record also take up some space...
 - 4 partition-table entries $\times$ 16 bytes per entry = 64 bytes
- Older style MBR format:
 - 512 bytes – 2 byte signature – 64 byte partition table = 446 bytes for bootloading
- Newer MBRs include additional details that reduce the MBR loader size to 434-436 bytes, broken into two parts
- Note: Advanced bootloaders like LILO and GRUB clearly can't fit in this small space
 - The bootloader itself is broken into multiple stages
 - Stage 1 is 512 bytes, responsible for loading stage 2 into memory
 - Stage 2 is much larger, e.g. 100s of KBs

Bootloading and MBRs (3)

- MBR loader must load a partition bootloader...
- Must do it in such a way that the partition bootloader *doesn't know* it wasn't loaded directly by the BIOS
 - MBR loader must emulate the BIOS bootloading mechanism
- MBR loaders use a mechanism called **chain loading**
- MBR loads the next bootloader into address 0x7C00, then jumps to that address and begins running it
 - Also retains other register values from BIOS, e.g. %d1 contains same value that BIOS passed to the MBR loader
- Of course, the MBR loader was loaded at 0x7C00...
 - MBR loader copies itself to another location, then jumps to the copy of itself, before it loads the partition boot sector
 - Chain loaders often copy themselves to address 0x0600

BIOS Library Functions

- At this early stage, bootloaders rely on BIOS functions to interact with the computer hardware
- Bootloaders have same issue that user applications have: They don't know the addresses of BIOS operations
- All BIOS functions are invoked via software interrupts
- Example: `int $0x10` is used for video services
 - Specify operation to perform in `%ah`
 - Other parameters are stored in other registers
 - Invoke `int $0x10` to perform the operation

- Example: Print a '*' character in teletype mode:

```
movb    $0x0e, %ah    # Use BIOS teletype function
movb    '*', %al
int     $0x10          # Invoke BIOS video service
```


BIOS Disk I/O Functions

- Interrupt 0x13 is used for low-level disk access
 - As before, **%ah** specifies the operation to perform
- Example: read a sector from the disk
 - **%ah** = 0x02
 - **%al** = number of sectors to read
 - **%cx** = cylinder (top 10 bits) + sector (bottom 6 bits)
 - **%dh** = head
 - **%dl** = disk drive to read
 - 0x00 = floppy disk 1, 0x01 = floppy disk 2, ...
 - 0x80 = hard disk 1, 0x81 = hard disk 2, ...
 - Loaded sector data is stored into **%es : %bx**
- This function uses CHS (cylinder-head-sector) addressing to access disk sectors
 - Currently limited to 1024 cylinders, 255 (or 256) heads, 63 sectors
 - At 512 bytes per sector, this is ~7.9 GiB ☹

BIOS Disk I/O Functions (2)

- Because of CHS addressing limitations, manufacturers added extended functions to the BIOS
 - These functions use Logical Block Addressing (LBA)
 - Every sector on disk has an absolute index; first sector = index 0
- Extended read-sector from disk
 - `%ah` = 0x42
 - `%dl` = disk drive to read
 - `%ds:%si` = segment:offset pointer to a Disk Address Packet
- Disk Address Packet is a simple data structure
 - `dap[0x00]` = size of packet, 16 bytes (0x10) for our purposes
 - `dap[0x01]` is unused, should be 0
 - `dap[0x02-0x03]` = number of 512-byte sectors to read
 - `dap[0x04-0x07]` = segment:offset of buffer to receive the data
 - `dap[0x08-0x0F]` = starting index of the sectors to read, using LBA
 - (Don't forget that IA32 is little-endian...)

IA32 Memory Addressing

- From last slide: extended read-sector from disk
 - `%ah` = 0x42
 - `%dl` = disk drive to read
 - `%ds : %si` = **segment:offset pointer** to a Disk Address Packet
- *What is that “segment:offset pointer” thingy?!*
- IA32 architecture supports several different memory addressing mechanisms
 - Some provide advanced ways for modern operating systems to manage memory
 - Others are required for backward compatibility
- Consequence: when your awesome hex-core Core i7 boots up, it's required to act exactly like an 8086

IA32 Real-Addressing Mode

- IA32 has a segmented memory model
 - What the “segments” mean depends on the memory addressing mode that the processor is in
- Initially, IA32 processors start out in real-addressing mode (a.k.a. “real mode”)
- Instruction pointer (**ip**), flags register (**flags**) are 16 bits
- Generally, only the 16-bit registers are used
 - General-purpose registers: **ax**, **bx**, **cx**, **dx**, **sp**, **bp**, **si**, **di**
 - Can also access either the low or high byte as well, e.g. **ah** or **al**
 - Segment registers: **cs**, **ds**, **ss**, and **es**
 - **cs** = code segment, **ds** = data segment, **ss** = stack segment, **es** = extra segment (also **fs** and **gs** if that isn't enough)
 - On 32-bit platforms, can use general-purpose 32-bit registers (!!!)
 - **eax**, **ebx**, **ecx**, **edx**, **esp**, **ebp**, **esi**, **edi**

IA32 Real-Addressing Mode (2)

- In real-addressing mode, the processor supports a 20-bit physical address space (1MiB)
- To generate this 20-bit address, the processor combines a 16-bit segment value and a 16-bit offset value:
 - $address = segment \ll 4 + offset$
 - The value in a segment register is called a **segment selector**
- Important details:
 - Each segment is 64KB in size
 - Each segment starts 16 bytes after the previous segment starts
 - **Segments overlap in physical memory!**
- Can have multiple real-mode addresses that refer to same physical location, e.g. A5C0:2DE1 and A5BF:2DF1

IA32 Real-Addressing Mode (3)

- Segment register used depends on operation being performed:
 - When the processor uses `%ip` to look up the next instruction, it automatically uses the code segment: `%cs:%ip`
 - When the processor uses `%sp` to push or pop a value on the stack, it uses the stack segment `%ss:%sp`
 - When an instruction moves data to/from memory, it uses the data segment `%ds`
- Can specify the segment register to use in many instructions
 - `movb %cs:(%si), %al`
 - Moves the byte at `Memory[%cs << 4 + %si]` into `%al`
 - `xchg %si, %ss:(%sp)`
 - Exchanges value in `%si` with word at `Memory[%ss << 4 + %sp]`
 - Note: just because you access `(%sp)` doesn't mean it will use `%ss`! If you intend to access the stack, safest to specify `%ss:(%sp)`.
 - If you know that `%ds == %ss`, no need to specify the segment to use

IA32 Real-Addressing Mode (4)

- Can move values into/out of segment registers
 - Except for moving into `%cs`, which is effectively a jump...
- Example: bootloaders generally must set up their own data and stack segments

```
sub %ax, %ax          # Set ax = 0
mov %ax, %ds          # Set ds and ss segments to 0
mov %ax, %ss
mov $0xf000, %sp      # Stack starts at 0000:f000
```

- *What would happen if an interrupt occurred between the last two instructions?*
 - Who knows what stack pointer the interrupt handler would use?!
 - Would trash some random memory location

IA32 Real-Addressing Mode (5)

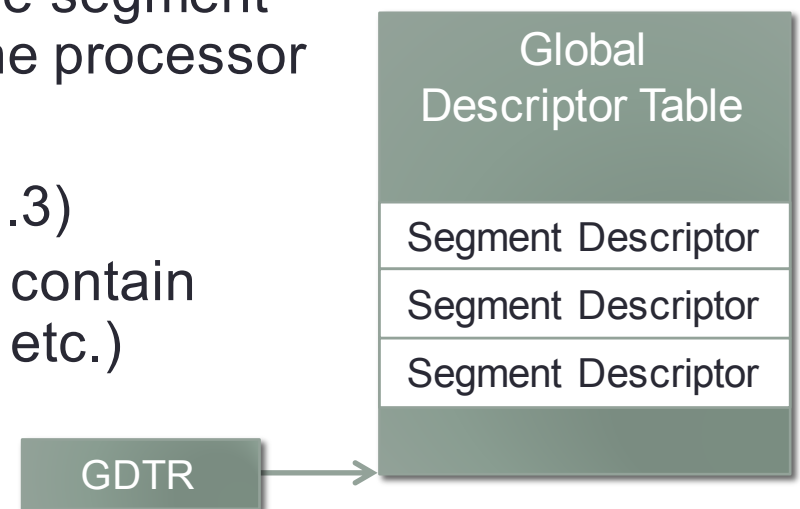
- When a value is moved into `%ss`, the CPU automatically disables interrupts until the next instruction executes
 - Allows a program to set the stack pointer in the next instruction
- Can also use `lss` instruction to set `%ss : %sp` in one step
- Note: in real-addressing mode, interrupt handlers are specified in a different way
 - An **interrupt vector table** specifies the segment selector and offset of each interrupt handler (4 bytes per interrupt)
 - This table starts at physical address 0 in memory
 - This can be changed via the Interrupt Descriptor Table Register, but this is not recommended!
- For interrupt n , CPU gets handler address from entry $n * 4$

IA32 Protected Mode

- IA32 also has **protected-mode** memory management
 - Includes memory protection facilities and optional paging/virtual memory management facilities
- Protected mode also uses the segment registers
 - And, they are still 16 bits! `%cs`, `%ds`, `%ss`, `%es`, `%fs`, `%gs`
 - Various operations still use them as before, e.g. instructions are loaded from `%cs:%eip`, stack manipulation uses `%ss:%esp`, ...
- In protected mode, segment selectors are not directly used to compute linear addresses!
 - Real-addressing mode and protected-mode behaviors are completely different
- Rather, segment selectors are indexes to **segment descriptors** into the Global Descriptor Table (GDT)
 - Describe each memory segment's extent, and its protection level

IA32 Protected Mode (2)

- The Global Descriptor Table is specified by the operating system to configure IA32 memory segmentation
 - Address of the Global Descriptor Table is specified in the Global Descriptor Table Register (GDTR)
 - Address is loaded into GDTR using (privileged) `lgdt` instruction
- Table holds segment descriptors describing various memory segments:
 - The base address (i.e. the start) of the segment in the 4GB linear address space of the processor
 - The size (i.e. extent) of the segment
 - The privilege level of the segment (0..3)
 - The type of the segment (e.g. does it contain executable code, can it be written to, etc.)
 - Other details of the segment

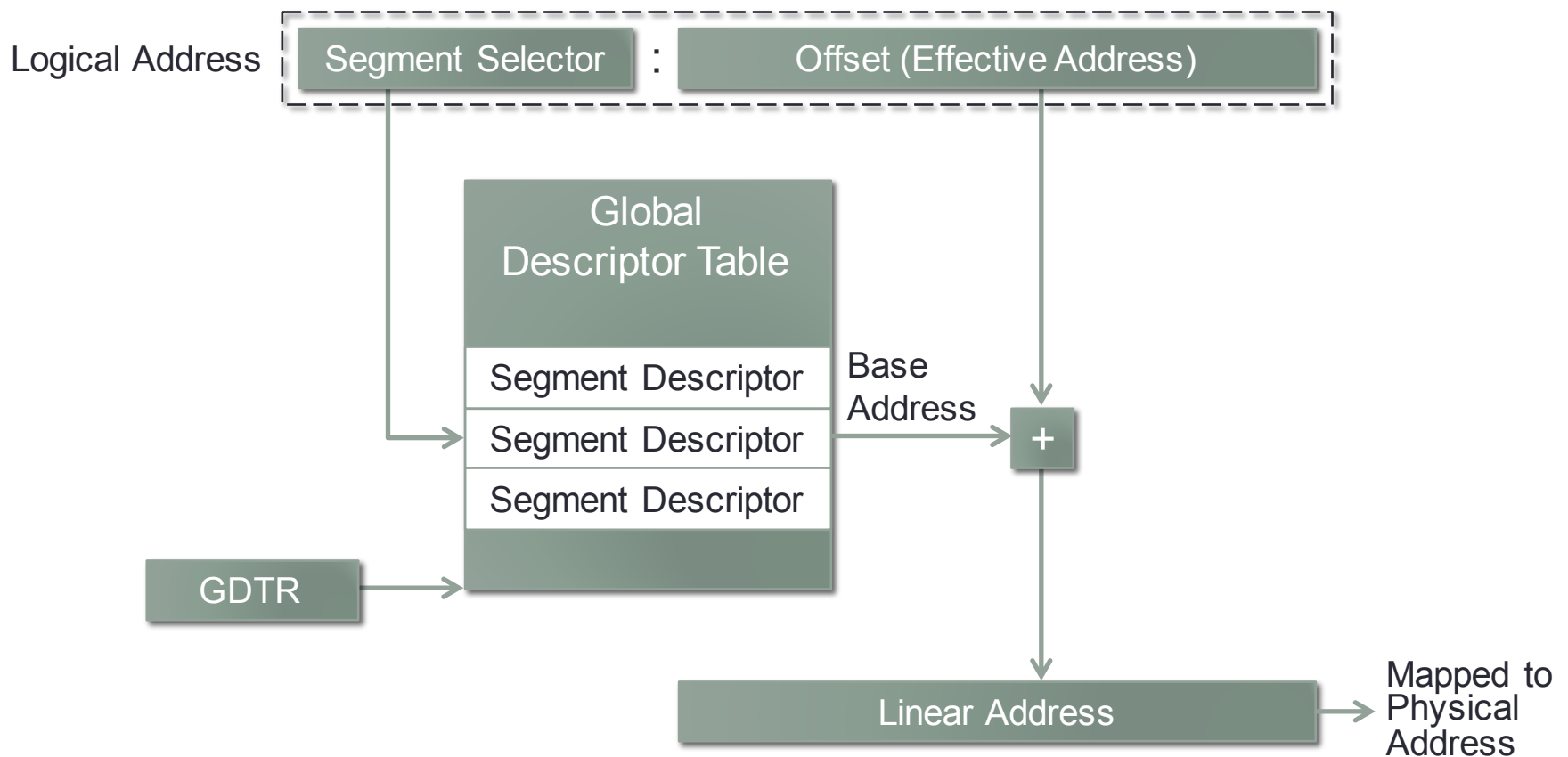


IA32 Protected Mode (3)

- All memory accesses in IA32 protected mode use **logical addresses**
 - *segment_selector : offset*, where *segment_selector* is 16 bits, and *offset* is 32 bits
 - The segment selector used is either taken from the segment register corresponding to the operation being performed...
 - Or, a segment selector can be specified in the instruction
- These logical addresses are mapped to **linear addresses** using the segment-descriptors in the descriptor table
 - $linear_address = gdt[segment_selector].base_address + offset$
 - The *offset* must be less than $gdt[segment_selector].size$, or else a general-protection fault will occur
- This linear address is a virtual address; it is then mapped to a **physical address** using the page-directory structure

IA32 Protected Mode (4)

- Pictorially:



- $linear_address = gdt[segment_selector].base_address + offset$

IA32 Protected Mode (5)

- Mapping from logical to linear addresses:
 - $linear_address = gdt[segment_selector].base_address + offset$
- This mechanism can be used to create very complex segmented memory models...
- Most operating systems expose a **flat memory model**
 - e.g. on 32-bit architecture, $base_address = 0x00000000$ and $size = 0xFFFFFFFF$
- Process isolation is enforced via the page table hierarchy and virtual memory manager
 - Each process is given its own page table hierarchy, mapping its linear addresses to physical addresses
- Note: On IA32, must still create two segments for kernel (one for code, other for data), and two segments for users
 - Segment descriptors specify a type, e.g. executable, writable, etc.

IA32 Protected Mode (6)

- One other place this is important: interrupt handling
- In protected mode, interrupts may run in a different protection ring than the code that was interrupted
 - Each protection ring has its own associated stack
 - Processor must switch stacks when the protection level changes
- The Task Register points to a structure that configures what stacks to use when switching protection levels
 - Structure is called a Task State Segment
 - Also refers to memory segments via Segment Selectors
- Protected-mode interrupt handling requires this data structure to be configured properly
 - Most operating systems make very limited use of the Task Register and task-state segments, so the configuration is straightforward

Next Time

- Finish discussion of IA32 bootstrap
- Unified Extensible Firmware Interface (UEFI)